# COMP462
# Project 3: State Estimation

In this project, we will implement the Particle Filtering algorithm for estimating the 2D pose of the robot base of a 7-DoF manipulator by letting the robot touch the surrounding obstacles in the environment. A code framework, hosted on github `https://github.com/robotpilab/COMP462/tree/main/project3`, is provided to facilitate the implementation.

**Report: A report is required for this project. The requirements of what to be included in the report are highlighted in the description.**

The simulation of the robot, which is based on `pybullet`, is provided. As illustrated in Figure 1, a stick with a small spherical end is fixed to the robot hand. To physically interact with and observe the environment, the robot needs to make contact with the surrounding obstacles by the spherical end of the stick.

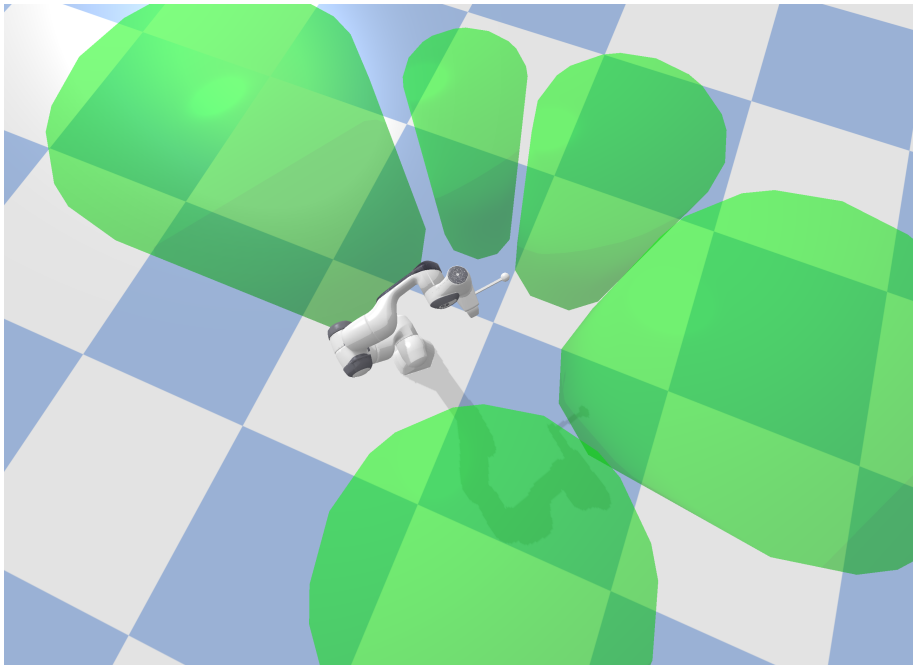

Figure 1: The simulation of the 7-DoF robot manipulator by `pybullet`. A stick with a spherical end is fixed to the hand of the robot. The robot will make contact with the surrounding cylinder obstacles (green) through the spherical end of the stick.

There are three parameters to estimate: $(x, y, \theta)$, where $(x, y)$ is the position of the robot base on the ground and $\theta$ is the orientation of the robot base about its $z$-axis (perpendicular to the ground). While the robot moves and touches the obstacles, each touch-based observation is represented by the robot configuration i.e., joint angles, at which the spherical end of the stick in the robot hand is in contact with an obstacle.

Before starting the project, please first run "`python join_urdf.py`" in the terminal to prepare the necessary URDF file for the simulation.

## Task 1: Particle Weight Calculation

In Particle Filtering, the state of a particle is represented by the system parameters to be estimated, and the weight of a particle represents the likelihood that the state of this particle matches the true

underlying values of the system. When a particle filter is in operation, the weights of particles are regularly updated everytime when some control or robot motion has been applied to the system, resulting in an updated observation.

Now, you will be given 10 particles, which represent 10 hypothesized robot base poses, and 1 observation that indicates the robot has made a contact with an obstacle with a joint configuration. You need to calculate the weights for each particle and find the most likely one. The major steps for this task are detailed below:

1. First, implement the following function in `alg.py` to obtain the distance from the stick's spherical end centered at $(x, y)$ to its closest cylinder obstacle:

```
def dist_to_closest_obs(x, y)
    """
    Find the distance between the stick's sphere end centered at (x, y)
    to its closest cylinder obstacle.
    args:       x: The x-coordinate of the spherical end.
                y: The y-coordinate of the spherical end.
    returns: dist: The distance to the closest obstacle.
    """
```

   In the context of this project, it is worth noting that the distance between a sphere and a cylinder should be defined by the distance between their (side) surfaces, not centers. Such distance can be calculated by the distance between their centers subtracting the sum of their radii. When the sphere contacts a cylinder, the distance should be zero; and when the sphere penetrates a cylinder, the distance should be negative; otherwise, a positive distance is expected. The closest cylinder obstacle is the one whose distance, either positive or negative, from the sphere is closest to zero.

   There are 5 cylinder obstacles in the environment. You can access the 2D positions and radii of them through the variables `OBS_CENTER` and `OBS_RADIUS` respectively. The radius of the stick's spherical end is in the variable `SPH_RADIUS` whose value is 0.02.

2. Next, you need to implement the following function in `alg.py` to evaluate the weights for the particles according to the given observation (i.e., the robot's joint angles when the spherical end of the stick is in contact with one of the cylinder obstacles):

```
def cal_weights(particles, obv, sigma=0.05)
    """
    Calculate the weights for particles based on the given observation.
    args:  particles: The particles represented by their states
                      Type: numpy.ndarray of shape (# of particles, 3)
                 obv: The given observation (the robot's joint angles).
                      Type: numpy.ndarray of shape (7,)
               sigma: The standard deviation of the Gaussian distribution
                      for calculating likelihood (default: 0.05).
    returns: weights: The weights of all particles.
                      Type: numpy.ndarray of shape (# of particles,)
    """
```

   Suppose the number of particles is $N$, then the input variable `particles` in the above function should be $(N, 3)$, of which each row represents the state of one particle. If we denote the state of the $i$-th particle (i.e., the $i$-th row of the variable `particles`) by $(x_i, y_i, \theta_i)$, then the weight of this particle should be calculated by:

$$w_i = \frac{\varphi(d_i)}{\sum_{j=1}^{N} \varphi(d_j)}$$

   where $\varphi(\cdot)$ is the probability density function (pdf) of a **zero-mean** Gaussian distribution with its standard deviation being `sigma` (whose value is 0.05 by default); The denominator

of the above equation is used to normalize the weights of the particles such that their sum is 1 (i.e., $\sum_{i=1}^{N} w_i = 1$). This is because the weights will be used as the probabilities for resampling particles in the Particle Filtering algorithm.

Assuming the robot base is at $(x_i, y_i, \theta_i)$, as represented by the $i-th$ particle , and the robot's joint angles are given by the observation (i.e., the input variable `obv`), $d_i$ in the above equation is the distance between the stick's spherical end and its closest cylinder obstacle (calculated by your implemented function in Step 1).

To obtain the 2D position (i.e., $x$ and $y$ coordinates) of the stick's spherical end with respect to the robot base frame when the robot's joint angles are `obv`, you need the forward kinematics function of the robot. You can use the provided forward kinematics solver to do this calculation by calling `FK_Solver.forward_kinematics_2d(obv)`.

3. Finally, please complete the following function in `alg.py` to find the index of the most likely particle (i.e., the particle with the highest weight):

```
def most_likely_particle(particles, weights):
    """
    Find the most likely particle.
    args:  particles: The particles represented by their states
                      Type: numpy.ndarray of shape (# of particles, 3)
                 obv: The given observation (the robot's joint angles).
                      Type: numpy.ndarray of shape (7,)
    returns:     idx: The index of the most likely particle
                      Type: int
    """
```

After completing this part, you can run the following command in your terminal to test your program: `python main.py --task 1`, and the results of running your implementation will be printed in the terminal.

**Report: With `sigma` = 0.05 (the standard deviation of the Gaussian), please report the weights of all particles and the index of the most likely particle. Similarly, by setting `sigma` = 0.1 and `sigma` = 0.5, repeat the experiment and report the particle weights and the index of the most likely particle. Provide your analysis of how the value of `sigma` would affect or not affect the particles.**

## Task 2: Estimation with Given Observations

In this part, you will be given 100 observations, that is, 100 different robot configurations where the robot is in contact with an obstacle. You need to implement the Particle Filtering algorithm by iteratively updating the particles and their weights with each observation. Please finish the `TODO` in the following function in `alg.py`:

```
def particle_filter(panda_sim, obvs, num_particles, sigma=0.05, delta=0.01)
    """
    The Particle Filtering algorithm.
    args:     panda_sim: The instance of the simulation.
                         Type: sim.PandaSim (provided)
                  obvs: The given observations (the robot's joint angles)
                         Type: numpy.ndarray of shape (# of observations, 7)
          num_particles: The number of particles.
                         Type: int
                  sigma: The standard deviation of the Gaussian distribution
                         for calculating likelihood (default: 0.05).
                  delta: The scale of the Gaussian for perturbing particles.
                         (default: 0.01)
    returns:        est: The estimate of the pose of the robot base.
```

```
                                    Type: numpy.ndarray of shape (3,)
        """
```

The particles are initialized by uniform random sampling in the space of the parameters - $(x, y, \theta)$, and the weight of each particle is equally initialized to be $\frac{1}{N}$ where $N$ is the number of the particles. Then, you need to complete the rest of the Particle Filtering algorithm.

At each iteration of the algorithm, you need to use one of the given observations (i.e., one row of `obvs`) to update the particles and their weights, and resample the particles according to their weights. Typically, the resampling process will keep the number of particles unchanged. As the particles are updated and resampled iteratively, they will finally converge to tightly cluster around the ground truth of $(x, y, \theta)$ of the robot base. Therefore, the estimate of $(x, y, \theta)$, which is calculated by the average of all particles, should be also close to the ground truth.

**Note:** It is crucial to slightly perturb each particle after the resampling in each iteration of Particle Filtering. To do this, for each particle $p_i = (x_i, y_i, \theta_i)$, you can add some Gaussian randomness by:

$$(x_i, y_i, \theta_i) + \epsilon_i$$

In the above expression, $\epsilon_i$ is a multivariate Gaussian random variable, i.e., $\epsilon \sim \mathcal{N}(0, \delta \mathbb{I}_{3\times 3})$, where $\delta$ (specified by the input argument `delta` of the function) is the scale of the Gaussian distribution, and $\mathbb{I}_{3\times 3}$ is the 3-by-3 identity matrix. This particle perturbation mechanism is essential for the robustness of Particle Filters. To see that, suppose no perturbation is applied and no randomness is added between iterations of the particle filtering algorithm. Then, when resampling particles, some particles with high weights can be sampled multiple times and the particles with low weights may not be sampled even once. As such, after resampling, there could be redundant particles and some other particles (especially those with low weights) could vanish. As the algorithm iterates, the number of different particles can gradually decrease to lose the distribution information of the parameters being estimated. To this end, by adding randomness between iterations, the particles can locally spread to effectively represent the distribution of the parameters, which is important to the robustness of the algorithm.

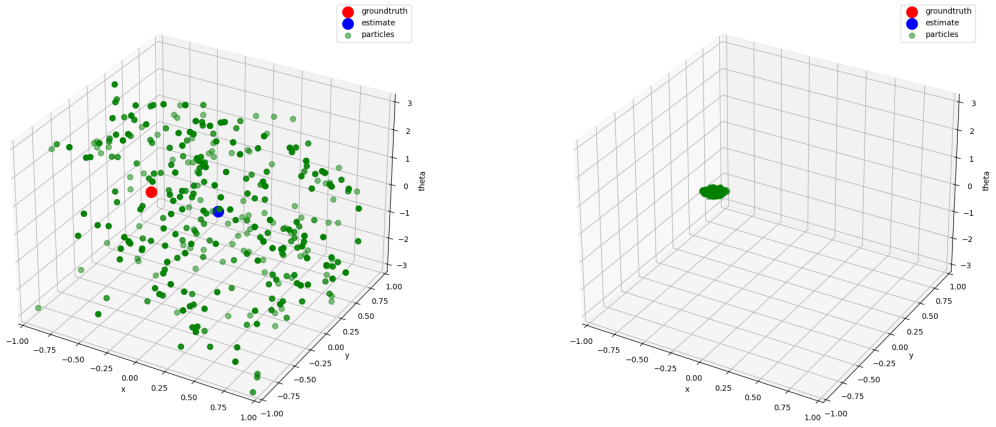


Figure 2: Visualization of the Particle Filtering algorithm in the parameter space, i.e., $(x, y, \theta)$. *Left:* At the beginning of Particle Filtering, the particles (green) are uniformly initialized, and the estimate (blue) of the parameters is not close to the groundtruth (red). *Right:* As the algorithm iterates, the particles will converge to the groundtruth.

After completing this part, you can run the following command in your terminal to test your program, which visualizes the robot configuration of each observation in the simulator and the particles in the parameter space shown in Figure 2: `python main.py --task 2`.

Moreover, to set the number of particles used in the algorithm, you can append `--num_particles the_number_of_particles` to the end of your command. And to set the standard deviation of

the Gaussian for updating weights (i.e., `sigma`) and the scale of the particle perturbations (i.e., `delta`), you can append `--sigma the_value_of_sigma` and `--delta the_value_of_delta` to the end of your command.

**Report: With 1000 and 2500 particles, and with `sigma` = 0.01, 0.05, 0.2 and `delta`=0.001, 0.01, 0.1 respectively, run your implemented Particle Filtering algorithm. For each combination of `sigma`, `delta` and the number of particles, you can run multiple trials. In the report:**

1. **discuss how the number of particles affect the performance of the Particle Filtering algorithm (e.g., the convergence speed, the robustness, etc)**
2. **discuss how the value of `sigma` affect the performance of the Particle Filtering algorithm (e.g., the convergence speed, the robustness, etc)**
3. **discuss how the value of `delta` affect the performance of the Particle Filtering algorithm (e.g., the convergence speed, the robustness, etc)**
4. **report your estimate of the pose, i.e., $(x, y, \theta)$ of the robot base**

## (Optional) Task 3: Estimation while Generating Observations

Instead of running your Particle filter with the provided observations we collected offline, in this part, you are asked to implement a method to collect observations online. For this task, it is recommended to complete the following steps:

1. First, please implement the following function in `alg.py` to control the robot manipulator in the simulation to collect an observation:

```
def get_one_obv(panda_sim)
    """
    Control the robot in simulation to obtain an observation.
    args: panda_sim: The instance of the simulation.
                     Type: sim.PandaSim (provided)
    returns:    obv: One observation found by this function.
                     Type: numpy.ndarray of shape (7,)
    """
```

   For this step, you are recommended to briefly read the codes of the provided class `PandaSim` in `sim.py` to get familiarized with the simulation environment, which is similar to the one in Project #1. Particularly, there are three functions in this class worth noting:

   The function `PandaSim.execute()` is used to command the robot by an input Cartesian velocity. The robot will move its end-effector with the velocity you specify by one simulation step (i.e., 0.002 seconds in simulation).

   The function `PandaSim.is_collision()` is used to detect collisions between the robot and the obstacles. This can be useful for safely commanding the robot to avoid unnatural behavior of the robot (e.g., penetration).

   The function `PandaSim.is_touch()` is used to detect if there is a contact between the stick's spherical end and any cylinder obstacle. You can use this function to determine when to obtain an observation.

2. Next, please complete the following function of online Particle Filtering in `alg.py`. The implementation of this function should be very similar to the one you completed in Task 2. The only difference is that here you need to integrate your implemented `get_one_obv()` function in Step 1 to use observations collected online.

```
def particle_filter_online(panda_sim, num_particles, sigma=0.05, delta=0.01)
    """
    The online Particle Filtering algorithm.
    args:     panda_sim: The instance of the simulation.
                         Type: sim.PandaSim (provided)
          num_particles: The number of particles.
                         Type: int
                  sigma: The standard deviation of the Gaussian distribution
                         for calculating likelihood (default: 0.05).
                  delta: The scale of the Gaussian for perturbing particles.
                         (default: 0.01)
    returns:        est: The estimate of the pose of the robot base.
                         Type: numpy.ndarray of shape (3,)
    """
```

After completing this part, you can run the following command in your terminal to test your program, which will render the simulation and visualize the particles in real time: `python main.py --task 3`.

**Report: In the report, please describe your online Particle Filtering algorithm (including how you control the robot and collect observations online) with both text and pseudo codes. Please visualize the particles in your report to show the convergence of the particles. You can choose to use any hyperparameters (the number of particles, the number of iterations, the values of `sigma` and `delta`) that you believe work the best. Please report the values of these hyperparameters you used in your experiments.**